Towards Certifying Domain-Specific Properties of Synthesized Code

Grigore Roşu
NASA Ames Research Center - USRA/RIACS
http://ase.arc.nasa.gov/grosu

Jonathan Whittle
NASA Ames Research Center - QSS Group Inc
http://ase.arc.nasa.gov/whittle

Abstract

In this paper, we present a technique for certifying domain-specific properties of code generated using program synthesis technology. Program synthesis is a maturing technology that generates code from a high-level specification in a particular domain. For acceptance of the generated code in safety-critical applications, it must be thoroughly tested (an estimated 80% of total software development costs in general). We show how the program synthesis system AUTOFILTER *can be extended to generate not only code but also proofs that properties hold in the code. We target properties that are sufficiently complex that they cannot be proved automatically from the code alone, but which can be proved using domain knowledge in the synthesis system producing proofs that can be easily checked by an independent verifier. This technique has the potential to reduce the costs associated with testing generated code.*

1. Introduction

Guaranteeing properties of code has long been a major goal of the Formal Methods community and has been tackled using static analysis, model checking and theorem proving methods. Of these, static analysis is perhaps the most practical and commercial systems such as PolySpace [13], which is based on abstract interpretation, are now emerging. Unfortunately, practical applications of static analysis techniques are limited to checking programming language level properties such as illegal type conversions, invalid arithmetic operations (e.g., division by zero) and overflow/underflow. Whilst important, such efforts do not address the issue of how to guarantee more complex properties and traditional techniques based on model checking and/or theorem proving are not yet viable.

Another growth area in the last couple of decades has been code generation. Although commercial code generators are mostly limited to generating stub codes from high level models (e.g., in UML), program synthesis systems that can generate fully executable code from high level behavioral specifications are rapidly maturing (see, for example, [16, 15]), in some cases to the point of commercialization (e.g., SciNapse [1]). In program synthesis, there is potential for automatically verifying more interesting properties because additional background information — from the specification and the synthesis knowledge base — is available. The claim made in this paper is that by coupling together program synthesis and property verification, it is possible to automatically certify that a piece of generated code satisfies certain complex properties. We illustrate this claim using a technique to certify properties of navigation software.

The motivation for this coupling is best illustrated by example. A common software development task in the spacecraft navigation domain is to design a system that can estimate the attitude of a spacecraft. This is typically mission-critical software because an accurate attitude estimate is necessary for the spacecraft controller to tilt the craft's solar panels towards the sun. Attitude estimators for different spacecraft are generally variations on a theme, and yet, currently, there is very little software reuse between projects. Program synthesis offers the potential to reduce development costs through rapid prototyping and a rapid turnaround cycle. However, for these kinds of applications, only 20% of effort is spent in software development, the other 80% being spent on validation of the code[1], including code walkthroughs, formal and informal testing. To reduce the 80% development costs, it is necessary to provide techniques that can avoid code inspections or reduce testing. We believe that property verification can reduce testing time and, moreover, that many properties cannot be verified automatically without the application of program synthesis. Verifying that a state estimator implementation *actually* produces a mathematically optimal estimate cannot be done automatically using the code alone, because the code does not reflect all the information needed, such as the statistical model. The use of additional domain knowledge and information from the synthesis process allows such verification tasks to be not only possible, but also automated.

Our approach to this problem is to *certify* crucial domain-specific properties in safety or mission critical do-

[1] Personal communication from the Jet Propulsion Laboratory (JPL).

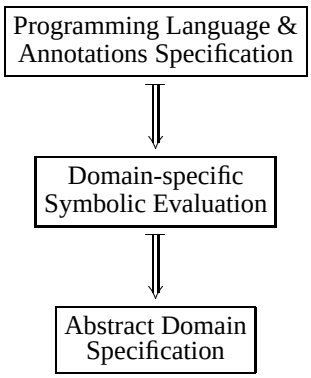

Figure 1. Domain-specific certifier.

mains. A domain-specific safety policy certifier was presented in [8], for the domain of coordinate frames which is also of crucial importance to astronomical navigation. Conceptually, a domain-specific certifier consists of three main components, as shown in Figure 1. The programming language and the abstract domain are linked via domain-specific abstract symbolic evaluation. Many software products are developed for domains that are quite complex and involve a significant body of mathematical knowledge, which is not the case for the semantics of standard programming languages in which these programs are usually written. Consequently, appropriate domain-specific *abstractions* of programming language constructs into domains of interest are required in order to perform domain-specific analysis of software. Unlike in standard static analysis of programs, where abstraction means grouping various values of the concrete domain into an abstract one (such as, each real number larger than 1 is "large positive"), in domain-specific analysis there might be no relationship between the concrete and the abstract domains; for example, 2.7 can be "meter", or "second", or "meter$*$Kg$*$second^{-2}", or any other measurement unit. One would, of course, like to certify software as automatically as possible, but this is very rarely feasible (due to intractability arguments) and clearly close to impossible for the complex domain presented in this paper. Therefore, user intervention is often needed to insert domain-specific knowledge into the programs to be certified, usually under the form of code annotations. The certifier in this paper needs annotations for model specifications, assertions, and proof scripts.

Our domain-specific certification approach requires more sophisticated reasoning than in approaches to date for *proof-carrying code (PCC)* [12]. The abstract domain specification is much richer than memory safety, and verifying the safety of each line of code can require tens of thousands of inference steps. The two specification levels, for the programming language and for the abstract domain, are independently reusable; e.g., once an abstract domain has been formulated it can be used to certify programs written in various programming languages, and conversely, programs can be certified for various domain-specific properties. *Extended Static Checker (ESC)* [5, 14] is a tool that finds programming errors at compile time, such as array index bounds errors, nil dereferences, deadlocks and race conditions. The user of ESC annotates the programs with specifications in a precondition-postcondition style (similar to ours) which are checked statically using a theorem prover for untyped predicate calculus with equality. The type system of the target programming language is implemented in untyped first-order logic. The use of ESC is therefore limited to programming language definable types and to properties that can be proved automatically using ESC's internal theorem prover. By allowing proof scripts as annotations in the programs to certify, we practically extend the usability of our certifiers to whatever properties that can be proved. However, some domain-specific proofs can be very complex, so, even if possible in theory, we do not anticipate that our domain-specific certifier will be used independently from the synthesis engine.

2. Domain-Specific Program Synthesis

Program synthesis is the generation of code from high-level, usually declarative, specifications of the expected behavior of the code. For safety-critical applications, a key concern when using synthesis is the correctness of the generated code. Traditional approaches to this problem use a theorem prover to derive a correctness proof and the code simultaneously. Unfortunately, such approaches rarely scale to realistic problems. Instead, successful synthesis systems (e.g., [1, 6]) favor a combination of advanced knowledge structuring mechanisms, search heuristics and symbolic solvers to generate (usually) domain-specific code, but the correctness of the code must be verified either by validating the synthesis system or through testing.

The key to making program synthesis successful is in choosing a domain in which the typical problems are sufficiently similar such that similar algorithms can be reused, but also sufficiently different as to make coding by hand non-trivial. The domain of state estimation is one example of these domains and is also an extremely widespread application area important to NASA and other major authorities such as the FAA.

AUTOFILTER is a program synthesizer for state estimation problems[2]. By state estimation, we mean estimating the state of an object (e.g., its position, attitude or noise

[2] AUTOFILTER is a redesigned, much extended version of Amphion/NAV [16], implemented in Prolog. Amphion/NAV was based on the SNARK theorem prover which was shown to be unsuitable for more complex problems in this domain and so was replaced by Prolog.

characteristics) based on noisy sensor measurements. This is an important problem found in spacecraft, aircraft and geophysical applications. The most common way of solving a state estimation problem is to use a recursive update algorithm known as the Kalman Filter [2] which provides a statistically optimal estimate of a state based on noisy sensor measurements. The Kalman Filter requires additional information to make this estimate, namely a model of the dynamics of the problem under study and a model of how the sensor measurements relate to the state:

$$x_{k+1} = \Phi_k x_k + w_k \quad (1)$$

$$z_k = H_k x_k + v_k \quad (2)$$

$$E[w_k] = E[v_k] = 0 \quad (3)$$

$$E[w_k w_i^\top] = \delta(k-i)Q_k \quad (4)$$

$$E[v_k v_i^\top] = \delta(k-i)R_k \quad (5)$$

x_k is a vector of state variables at time k. In a typical attitude estimation problem, for example, the state vector, x_k, might contain three variables representing rotation angles of a spacecraft. This is what will the Kalman Filter will estimate. Equation (1) is the process model which describes the dynamics of the the state over time[3] — the state at time $k+1$ is obtained by multiplying the state transition matrix Φ_k by the previous state x_k. The model is imperfect, however, as represented by the addition of the process noise vector w_k. Equation (2) is the measurement model and models the relationship between the measurements and the state. This is necessary because the state often cannot be measured directly. The measurement vector, z_k, is related to the state by matrix H_k. v_k represents the noise in this relationship. Simplifying Kalman Filter assumptions state that all noises must be gaussian processes with zero mean and there must be no correlation between the noise over time (see (4) and (5) where $a^\top$ is the transpose of vector a and $\delta(j)$ is the Kronecker delta function which evaluates to 1 when $j = 0$ and 0 otherwise. Q_k and R_k are matrices which represent the noise characteristics of the process model noise and measurement model noise, respectively).

Given models of this form, a Kalman Filter can be implemented that optimally estimates the state vector x_k. A schematic algorithm for this Kalman Filter is given in Figure 2. The estimate, $\hat{x}_k$, can be proved to be an optimal estimate of the state, x_k, in the sense that the mean squared error (also known as the error covariance matrix), $E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^\top]$, is minimized (see the next section). Any Kalman Filter should satisfy this minimization property. In Figure 2, P_k is the error covariance matrix and is updated on each iteration of the filter. P_k gives an indication of the error in the filter estimates and so is used as a check whether or not the filter is converging.

[3] the formulation given here is a discrete one

As an example of how Kalman Filters work in practice, consider a simple spacecraft attitude estimation problem. Attitude is usually measured using gyroscopes, but the performance of gyroscopes degrades over time so the error in the gyroscopes is corrected using other measurements, e.g., from a startracker. In this formulation, the process equation (1) would model how the gyroscopes degrade and the measurement equation (2) would model the relationship between the startracker measurements and the three rotation angles that form the state (in this case, H_k would be the identity matrix because startrackers measure rotation angles directly). From these models, a Kalman Filter implementation would produce an optimal estimate of the current attitude of the spacecraft, where the uncertainties in the problem (gyro degradation, startracker noise, etc.) have been minimized.

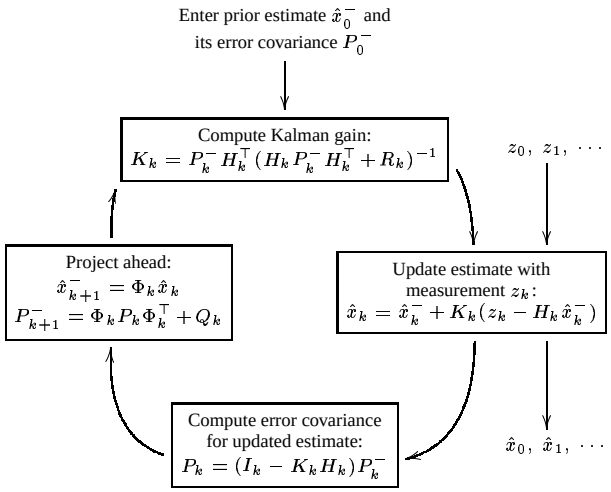

Figure 2. Kalman Filter Loop

AUTOFILTER takes as input a mathematical specification including equations (1) - (5) but also descriptions of the noise characteristics and filter parameters. From this specification, it generates code that implements (some variant of) the algorithm in Figure 2. In fact, AUTOFILTER generates code in our own intermediate language which is then translated into C++ or Matlab. In this paper, we only consider code in the intermediate language (see Figure 3 for an example of its syntax). It should be noted that Figure 2 represents just one of many possible variations and configurations of the filter. In fact, the abundance of variations is what makes this domain ideal for synthesis. Examples of other variations are the extended kalman filter, the information filter, the unscented filter.

```
1.  input xhatmin(0), pminus(0);
2.  for(k,0,n) {
3.    gain(k) := pminus(k) * mtrans(h(k)) * minv(h(k) * pminus(k) * mtrans(h(k)) + r(k));
4.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - (h(k) * xhatmin(k))));
5.    p(k) := (id(n) - gain(k) * h(k)) * pminus(k);
6.    xhatmin(k + 1) := phi(k) * xhat(k);
7.    pminus(k + 1)  := phi(k) * (p(k) * mtrans(phi(k))) + q(k);
8.  }
```

Figure 3. Kalman Filter code calculating the best estimate incrementally.

3. An Informal Optimality Proof

In this paper, we describe techniques for certifying domain-specific properties of code generated by AUTOFILTER. In particular, we consider the optimality proof introduced in the previous section — of the minimization of the mean squared error. Despite the apparent simplicity of the code in Figure 3 that AUTOFILTER generates, the proof of optimality is quite complex. The main proof task is to show that the vector `xhat(k)` (corresponding to $\hat{x}_k$ in the previous section) is the best estimate, under simplifying assumptions, of the state vector x_k at time k. This is a standard proof in state estimation and is usually presented in books as an informal mathematical proof several pages long. We sketch the proof in this section, emphasizing those aspects which are particularly relevant for automating the proof, especially the *assumptions*.

The very first assumption made in all books is that the initial estimates, $\hat{x}_0^-$ and P_0^-, are the best prior estimate and its error covariance matrix (that is, $E[(x_0 - \hat{x}_0^-)(x_0 - \hat{x}_0^-)^\top]$), respectively. At a given time k, if $\hat{x}_k^-$ is the best prior estimate then one can use equation (2) to conclude that the most probable measurement error is $z_k - H_k\hat{x}_k^-$. Another assumption is that "the best estimate is a linear combination of the best prior estimate and the measurement error". Formally, this says that the best estimate $\hat{x}_k$ is somewhere in the image of the function $\hat{x}_k(y) := \lambda y.(\hat{x}_k^- + y * (z_k - H_k\hat{x}_k^-))$, where the coefficient y is a matrix having as many rows as $\hat{x}_k^-$ and as many columns as rows z_k. We are looking for the y corresponding to the minimum error covariance matrix, $P_k(y) := E[(x_k - \hat{x}_k(y))(x_k - \hat{x}_k(y))^\top]$, that is, the solution of the derivative of $P_k(y)$ with respect to y[4]. In fact, differentiation of matrix functions is a complex field that we partially formalized and which we cannot cover here, but it is worth mentioning, in order for the reader to anticipate the non-triviality of this proof, that the y giving the minimum of $P_k(y)$ is the solution of the equation $d(trace(P_k(y)))/dy = 0$, where the trace of a matrix is the sum of the elements on its first diagonal and for a (standard) function $f(y_{11}, y_{12}, ...)$ on the elements of a matrix y, its derivative df/dy is the matrix $(df/dy_{ij} \mid y_{ij} \in y)$ having the same dimensions as y. Assuming that P_k^- is the error covariance of the best *prior* estimate of $\hat{x}_k$, that is, $E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^\top]$, then after calculations (which formally involve several thousands of uses of basic properties of matrices and differentiation as argued in Section 5) one gets the solution

$$K_k := P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1},$$

which is what line 3 of our program in Figure 3 calculates. One can also calculate the best estimate now, namely $\hat{x}_k(K)$, which is what line 4 of our program does, and also the error covariance matrix of the best estimate, namely $P_k(K)$, which is what line 5 does; notice that the latter calculations also take several thousands basic proof steps.

In order to complete the proof, one needs to show that $\hat{x}_{k+1}^-$ and P_{k+1}^- are the best prior estimate and its error covariance matrix at time $k+1$, respectively. The former follows by another unanimously accepted assumption among experts, namely that the best prior estimate at the next step follows the state equation (1) using the best estimate at the current state, but where the noise is *ignored*; the intuition for this assumption is that the current best estimate is random anyway, so the noise with mean 0 can be ignored. We do this in line 6. The latter can be also obtained by calculations, also taking several thousand basic proof steps, transforming the expression $P_{k+1}^- = E[(x_{k+1} - \hat{x}_{k+1}^-)(x_{k+1} - \hat{x}_{k+1}^-)^\top]$ by replacing $\hat{x}_{k+1}$ as in equation (1) and $\hat{x}_{k+1}^-$ as in line 6.

4. A Framework for Formalizing the Proof

To generate and automatically certify proofs such as the one given in the previous section, we need to formalize the domain knowledge, which includes matrices, functions on matrices, and differentiation. We first discuss the formal language that we chose for this purpose together with its underlying logic. The reader can download our abstract domain formalization from `http://ase.arc.nasa.gov/grosu/download/kalman`.

[4]Technically speaking, one should also take the second derivative to show that the solution is indeed a minimum, but this step is considered "obvious" and constantly skipped by experts.

4.1. Maude and Membership Equational Logic

Maude [4] is a freely distributed high-performance executable specification system in the OBJ [7] family, supporting both rewriting logic [9] and membership equational logic [10]. Because of its efficient rewriting engine and because of its metalanguage and modularization features, Maude is an excellent tool to develop executable environments for various logics, models of computation, theorem provers, and even programming languages.

Membership equational logic (MEL) [10] is a variant of equational logic which, in addition to atomic equalities $t = t'$, allows atomic *memberships* $t : s$ stating that the term t has the sort s. In Maude, conditional equations and memberships are declared with the keywords `ceq` and `cmb`, respectively, while the unconditional ones with `eq` and `mb`. For example, the conditional membership `cmb X/Y : Real if Y=/=0` states that for any reals `X` and `Y`, `X/Y` is a real, or has the sort `Real`, if `Y` is non-zero. Sorts are grouped in *kinds* and operations are defined only on kinds, but Maude provides convenient syntactic sugar conventions. For example, a subsort declaration `subsort Nat < Real` is syntactic sugar for the membership `cmb X : Real if X : Nat`, the operation declaration `op _+_ : Real Real -> Real` is syntactic sugar for[5] `cmb X+Y : Real if X : Real and Y : Real`, and `[Real]` is a shorthand for the kind containing the sort `Real`; the operation declaration `_/_ : Real Real -> [Real]` says that a quotient term might not have a sort, i.e., it might be *undefined* in a partial algebraic terminology. There is an automatic translation from partial equational logic, or more precisely from its more general variant called *partial membership equational logic (PMEL)*, into MEL explored in detail in [10, 11]. Maude's *implicit* support for partiality was a major factor in choosing Maude as a logic and implementation engine for our certification tools, because all the specifications of abstract domains that we encountered so far involve partial operators. However, we warn the reader that Maude does *not* explicitly support PMEL, i.e., it is a *total* MEL engine[6]. That means that, for example if one declares the equation `eq X/(Y/Z) = (X*Z)/Y` then one should not expect Maude to implicitly prove that `Y/Z` is defined before applying the equation; it is the user's responsibility to test this, i.e., to use an equation of the form `ceq X/(Y/Z) = (X*Z)/Y if Y/Z : Real`. Since we use partial specifications often and since it is so easy to omit such membership checks, we have developed an automatic criterion that checks whether a MEL theory is *duplex*, i.e., if it can be safely regarded as a specification in PMEL [11]. Our 200 axiom abstract domain presented next passed the duplex criterion, so we know that Maude's total reasoning is sound for our domain.

[5] Together with an appropriate operation declaration on kinds.

[6] We are not aware of any tool providing explicit executional support for partial equational logics.

4.2. Matrices

The main problem in automating equational proofs is that equational axioms can be used both forwards and backwards, so rewriting alone is not sufficient and search can quickly become intractable. Equational reasoning with matrices is extensively used in all state estimation optimality proofs that we are aware of. In fact, most of the proof steps in our scripts and all our lemmas are equational. What is less obvious is that most of the operations and axioms/lemmas in matrix theory are *partial*. For example, addition is defined iff the two matrices have the same dimensions, multiplication is defined iff the number of columns of the first matrix equals the number of rows of the second, and the commutativity and associativity of addition hold true iff the matrices involved have the same dimensions. It was a big benefit, if not the biggest, that Maude provided support for partiality, thus allowing us to compactly specify matrix theory and do partial proofs. The partial infix operation of multiplication and the total transpose operation are defined as follows:

```
op _*_ : Matrix Matrix -> [Matrix] .
op mtrans : Matrix -> Matrix .
```

In order to define their semantics and properties, we need two (total) operations that give the numbers of columns and rows of a matrix that we denote `c` and `r`, respectively, of arity `Matrix -> Nat`. Now we can express definedness of multiplication together with appropriate conditional equations computing the new columns and rows:

```
cmb P * Q : Matrix if c(P) == r(Q) .
ceq c(P * Q) = c(Q) if P * Q : Matrix .
ceq r(P * Q) = r(P) if P * Q : Matrix .
```

Axioms relating various operators on matrices are also needed, such as:

```
ceq mtrans(P * Q) = mtrans(Q) * mtrans(P)
   if P * Q : Matrix .
```

together with more than 50 others, most of them conditional and involving memberships.

4.3. Functions on Matrices

One step in optimality proofs is stating that the best estimate of the actual state is a linear combination of the best prior estimate and the measurement error. The coefficients of this linear dependency is calculated such that the error covariance matrix is minimized. Therefore, before the optimal coefficient is calculated, and in order to calculate it, the best estimate vector is regarded as a *function* of the form $\lambda y.(\langle \textit{prior estimate} \rangle + y * \langle \textit{measurement error} \rangle)$. In order for this function to be well defined, y must be a matrix having appropriate dimensions as given in Section 3.

Hence, we need to formally define functions on matrices together with their properties. We do it by declaring new sorts, `MatrixVar` and `MatrixFun`, the first being a subsort of `Matrix`, together with operations for defining functions and for applying them, respectively:

```
op /\_._ : MatrixVar Matrix -> MatrixFun .
op _ _ : MatrixFun Matrix -> [Matrix] .
```

Appropriate (conditional) axioms for functions are specified, also taking into account partiality, such as:

```
ceq (/\y.(P+Q))(X) = (/\y.P)(X) + (/\y.Q)(X)
   if P+Q : Matrix .
ceq (/\y.y)(X) = X
   if c(X) == c(y) and r(X) == r(y) .
```

among many others.

4.4. Differentiation

If P is a square matrix then *trace*(P) is the sum of all P's elements on the main diagonal. Axiomatization of functions on matrices with their derivatives can be arbitrarily complicated; our approach is top-down, i.e., we first define properties *by need*, use them, and then prove them from more basic properties when possible. For example, the only property that we used so far linking optimality of estimates to differentiation is that a matrix K minimizes a function $\lambda y.P$ iff $(d(trace(\lambda y.P))/dy)(K) = 0$. For that reason, in order to avoid going into deep axiomatizability of mathematics, we have just defined a "derived" operation

```
op d(trace_)/d_ : MatrixFun MatrixVar
  -> MatrixFun .
```

which gives directly the derivative of the trace of a function, and declared some properties of it, such as the conditional equation

```
ceq d(trace(/\Y.(Y*P)))/d(Y) = /\Y.mtrans(P)
   if Y*P : Matrix and r(Y) == c(P) .
```

stating that the derivative of the function $\lambda y.(y * P)$ is $\lambda y.P^\top$ whenever $y * P$ is a well-defined square matrix. One could, of course, prove this property from more basic properties of traces, functions and differentiations, but one would need to add a significant body of mathematical knowledge to the system. We will eventually do it when our synthesis and certification systems become more stable, but for now we prefer to just give these provable properties as axioms of the abstract domain.

5. A Formal Optimality Proof

In this section we explain how we formalized the informal proof in Section 3, using the axiomatization of the abstract domain in Section 4. This formalization was done manually, using an interactive theorem prover implemented in Maude and presented below. This proof was a painful and time consuming task, not only because of its mathematical complexity, but also because the axiomatization of the abstract domain changed often as we understood the domain and the requirements for our proofs better. We are currently working on another proof of optimality, for information filters [2], which follows the same pattern. One of the major benefits of synthesizing certifiably correct code is that such proofs will be done by trained experts *only once*, stored in a generic form in the synthesis engine, and then reused/instantiated many times in the generated annotated programs.

5.1. The ITP Tool

The ITP tool [3] is an experimental interactive inductive theorem prover implemented by metalevel programming and rewriting in Maude. The input of ITP is a pair `specification |- sentence`, called a *goal*. If the sentence can be automatically proved by applying the equalities in the specification as rewriting rules then the user gets the desired `q.e.d.` message; otherwise, some simplified form of the sentence to be proved is returned and the user is expected to provide hints, such as for example `apply -distr to 1.2 at 2.2.3` which says that the distributivity axiom should be applied backwards to proof task number `1.2` at position `2.2.3`. Simplifications by rewriting are automatically done after each hint. We have used hundreds of hints in our optimality proofs presented next. Many new proof tasks can be generated during a proof due to lemma introduction and/or induction. There are rigorous conventions in ITP for labelling the axioms and the proof tasks, and also in accessing positions in terms, but these are not important for this presentation and may change in the near future as ITP improves, so we omit them.

5.2. Specifying the Statistical Model

In order to reason about the code in Figure 3, one must know where the matrices `z`, `h`, etc., come from and what is their abstract meaning, or in other words, one needs the *specification* of this particular Kalman Filter together with all its *assumptions*. These are needed in *addition* to the abstract domain knowledge in Section 4. Therefore, the very first step is to expand the abstract domain with this Kalman Filter's specification, that we denote SPEC_{KF}, which declares all the matrices and vectors involved together with their dimensions, such as

```
ops x phi w : MachineInt -> Matrix .
ops z h v : MachineInt -> Matrix .
ops r q : MachineInt -> Matrix .
ops n m k : -> MachineInt .
var K : MachineInt .
eq r(x(K)) = n .    eq c(x(K)) = 1 .
eq r(phi(K)) = n .  eq c(phi(K)) = n .
```

```
eq r(w(K)) = n .    eq c(w(K)) = 1 .
eq r(z(K)) = m .    eq c(z(K)) = 1 .
eq r(h(K)) = m .    eq c(h(K)) = n .
eq r(v(K)) = m .    eq c(v(K)) = 1 .
...
```

as well as model equations/assumptions, such as

```
eq x(K + 1) = (phi(K) * x(K)) + w(K) .
eq z(K) = (h(K) * x(K)) + v(K) .
eq r(K) = E[v(K) * mtrans(v(K))] .
eq q(K) = E[w(K) * mtrans(w(K))] .
...
```

Other axioms/assumptions that we do not formalize here due to space limitation include independence of noise, and the fact that the best prior estimate at time `k + 1` is the product between `phi(k)` and the best estimate calculated previously at step `k`. One major problem that we encountered while developing our proofs was that these and other assumptions not mentioned here are so well (and easily) accepted by experts that they don't even make their use explicit in their informal proofs; this was of course unavoidable in the context of formal proving, so we had to declare them explicitly as axioms.

This specification has about 35 axioms/assumptions and the interested reader can download it from the URL `http://ase.arc.nasa.gov/grosu/download/kalman`. A major advantage of our approach to combine synthesis and certification is that specifications can be generated *automatically* from the problem description input to the synthesis engine.

5.3. Modularizing the Proof

In order to machine check the proof of optimality, the proof must be decomposed and linked to the actual code. This is done by adding the specification above at the beginning of the code and adding appropriate formal statements, or assertions, as annotations between instructions, so that one can prove the next assertion from the previous ones and the previous code. Proofs are also added as annotations where needed. Notice that by "proof" we here mean a series of hints that ITP uses to guide the proof. The resulting annotated code is shown in Figure 4, where we replaced the more formal (and longer) assertions by English. The proof assertions in Figure 4 should be read as follows: proof assertion n is a proof of assertion n in its current environment.

The best we can assert between instructions 1 and 2 is that `xhatmin(0)` and `pminus(0)` are initially the best prior estimate and error covariance matrix, respectively. This assertion is an assumption in SPEC_{KF} and so can be immediately checked.

Between 2 and 3 we assert that `xhatmin(k)` and `pminus(k)` are the best prior estimate and error covariance matrix, respectively. This is obvious for the first iteration of the loop, but needs to be proved for the other iterations. Therefore, we do an implicit proof by induction.

The assertion after line 3 is that `gain(k)` minimizes the covariance matrix of the error between the real (unknown) state of the system and a linear combination of the best prior estimate and our current measurement error. This formal assertion is rather technical and takes a few lines of Maude code, so it is not worth showing it here. It was, however, the part of the proof that was the most difficult to formalize. Its proof script contains 7 lemmas and it has 142 steps, that is, there are 142 uses of labeled (conditional) sentences. This means that there are at least 142 places where an automatic equational theorem prover would try both directions of an equation — 2^{142} combinations. These numbers make us strongly believe that there is almost no chance to get these proofs automatically. Note that there are many many more unlabeled rewritings applied in between, most of them to prove the conditions of others. Taking into account the 15 seconds that ITP spent to check the 142 lines of proof script, the speed of Maude (3 million rewrites per second) and pessimistically assuming that its ITP slows it down 1000 times, then we predict that there are at least 45,000 uses of the axioms in the abstract domain and SPEC_{KF} in this proof.

The assertion between lines 4 and 5 says that `xhat(k)` is the best estimate of the actual state and follows now immediately from the previous assertion. After line 5, however, we have the assertion that `p(k)` is the error covariance matrix of the best estimate and its proof needs 4 lemmas and has about 110 proof script steps. After line 6, due to an assumption in SPEC_{KF}, we can assert and easily show that `xhatmin(k+1)` is the best prior estimate at time `k+1`, which together with the instruction on line 7 implies the assertion between lines 2 and 3, so we have completed our proof of optimality of the code in Figure 3 by induction. It took ITP a bit more than 30 seconds to check all this proof, which makes us predict at least 100,000 of uses of axioms.

6. Synthesizing Annotated Kalman Filters

AUTOFILTER synthesizes code by exhaustive, layered application of *schemas*. A schema is a program template with open slots and a set of applicability conditions. The slots are filled in with code fragments by the synthesis system calling the schemas recursively. The conditions constrain how the slots can be filled — they must be proven to hold in the given specification before the schema can be applied. Some of the schemas contain calls to symbolic equation solvers, others contain entire skeletons of statistical or numerical algorithms. By recursively invoking schemas and composing the resulting code fragments, AUTOFILTER is able to automatically synthesize programs of considerable size and internal complexity.

Figure 5 gives an abstraction of a top-level schema for

```
  /* Specification of the state estimation problem
        ... about 35 axioms/assumptions in Maude */
1.  input xhatmin(0), pminus(0);
  /* Assertion 1: (in English)
        xhatmin(0) and pminus(0) are the best prior estimate and its error covariance matrix */
2.  for(k,0,n) {
  /* Assertion 2: (in English)
        xhatmin(k) and pminus(k) are the best prior estimate and its error covariance matrix */
3.    gain(k) := pminus(k) * mtrans(h(k)) * minv(h(k) * pminus(k) * mtrans(h(k)) + r(k));
  /* Proof assertion 3: (142 ITP hints including those below)
        (lem (l(pminus(k))) = (n) to (1) .)
        (apply assertion-1-2 to (1 . 0 . 1) at (1) .)
        ... the 138 other hints are omitted ...
        (apply  pminuskmtrans to (1) at (1 . 2 . 1 . 1 . 1) .)
        (apply  comm+ to (1) at (1 . 2) .) */
  /* Assertion 3: (in English)
        gain(k) minimizes the error covariance matrix */
4.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - (h(k) * xhatmin(k))));
  /* Proof assertion 4: (... omitted) */
  /* Assertion 4: (the main goal)
        xhat(k) is the best estimate */
5.    p(k) := (id(n) - gain(k) * h(k)) * pminus(k);
  /* Proof assertion 5: (... omitted; 110 ITP hints) */
  /* Assertion 5:
        p(k) error covariance matrix of xhat(k) */
6.    xhatmin(k + 1) := phi(k) * xhat(k);
  /* Proof assertion 6: (... omitted) */
  /* Assertion 6:
        xhatmin(k + 1) best prior estimate at time k + 1 */
7.    pminus(k + 1)  := phi(k) * (p(k) * mtrans(phi(k))) + q(k);
  /* Proof assertion 2: (... omitted; 31 ITP hints) */
8.  }
```

Figure 4. Annotated Kalman Filter code calculating the best estimate.

generating Kalman Filter code. $name(\%)$ are schema slots. They are filled in by an assignment of the form $\%name :=$ .... In some cases, the top-level schema will fill slots directly. In other cases, the slots are filled by recursively invoking other schemas. The numbers in square parentheses refer to line numbers in Figure 3. The top-level schema generates a template for each line in Figure 3 which is filled in by recursively invoking other schemas. Note that this can result in code that is substantially different from that given in Figure 3 but still implements (some variant of) a Kalman Filter. For example, the slot for `propagateEstimate` would be filled in differently for a standard Kalman Filter than for an information filter or an extended Kalman Filter. Slots in Figure 5 without associated line numbers are not important for the purposes of this paper but generally are concerned with book-keeping tasks such as storing the estimates in an output vector, updating time-varying matrices on each iteration, etc.

The schema recursively invokes other schemas to fill in its slots, e.g., to generate the `propagateEstimate` code. For certification purposes, the schema must also generate a *proof* that the schema is correct — in this case, the optimality proof in Figure 4. In order to generate this proof, the schemas are extended to generate also the assertions and proof assertions from Figure 4. In this way, each line of code generated comes optionally with an assertion and/or the proof that the assertion holds. The mechanics of adding the proofs to the schemas is mostly trivial. Each line of code generated is, in our intermediate language, of the form *code_fragment(Code, Attributes)* where $Code$ is the actual code and $Attributes$ is a list of artifacts that is generated along with the code. Attributes can be comments that explain the line of code or they can be assertions or proof assertions. Currently, the proof assertions in the attributes are ITP proof scripts, i.e., a sequence of applications of axioms/lemmas (along with variable substitutions). These proofs typically are very complex and involve the exploration of a large search space. The hints in the high-level proof scripts always occur at a choice point in the proof. Hence, given the proof script, it is possible to reconstruct the entire proof without the need for any search. The code in Figure 4 containing all the annotations can be found at `http://ase.arc.nasa.gov/grosu/download/kalman`. The reconstruction of the entire proof is currently done in the certifier, but it could just as easily have been given explicitly in the schema. An approach that we intend to investigate soon is to store these proof scripts together with the schemas in the AUTOFILTER knowledge base. The full proof can

```
/* applicability conditions */
 ... process noise is Gaussian
 ... measurement noise is Gaussian
 ... process/measurement noise are independent
                ...
/* set up template */
result := kalman(local(%),
                 initialize(%),
                 loop(%),
                 postloop(%))),
                ...
%loop := for(pvar,0,n,                      // [2]
             update(zupdate(%),
                    phiupdate(%),
                    hupdate(%),
                    gain(%),                // [3]
                    estimateUpdate(%),      // [4]
                    covarUpdate(%),         // [5]
                    storeOutput(%),
                    propagateEstimate(%), // [6]
                    propagateCovar(%)))     // [7]
/* fill in some slots */
                ...
/* recursively invoke schemas to fill
   remaining slots */
                ...
```

Figure 5. (Part of) a Kalman Filter schema

then be reconstructed at synthesis time and the entire proof can be added to the code (rather than just the ITP hints). This would allow the certifier to be as minimal as possible — it would merely be a *proof checker* that can be verified rather than a more complex hint interpreter as now.

7. Certifying Annotated Kalman Filters

There are various types and levels of certification, including testing and human code review. In this paper we address certifying conformance of programs to domain-specific properties. The general problem is known to be intractable, but by using program synthesis to annotate code with assertions and proof scripts, complex properties can be certified automatically. Our long term goal is to develop an automated state estimation certifier which:

- is simple, so that it can be easily validated by ordinary code reviewers;
- is general, so it works on a large variety of programs;
- reduces the amount of domain-specific knowledge to be trusted to a few easily readable properties, so that it can be validated by domain experts;
- is independent from the domain-specific synthesis system, so that the likelihood that the two systems have common abstract domain errors is minimized and therefore can be safely used together.

There are sensible trade-offs between these desired features. For example, if the certifier uses a specialized theorem prover then the synthesis engine can generate fewer and simpler annotations, but the certifier is itself complex and the certification process can take a long time. On the other hand, if the certifier is a simple proof checker then certification can be done relatively quickly and can be more easily accepted even by skeptical users, but one needs to generate very detailed proofs of correctness together with the code. The certifier described next uses a combination of theorem proving and proof checking.

Our current certifier takes as input a PMEL specification of the abstract domain and an annotated program and returns “yes” or “no”. It extracts proof tasks together with their proof scripts from the annotated program, and then calls Maude’s ITP tool to validate them. The proof tasks are generated from both annotations and code, while the proof scripts are extracted from annotations. It answers “yes” if and only if all the proof scripts are valid for the proof tasks that it generates. Therefore, like in proof carrying code, the code, the assertions and the proofs are interdependent, so one cannot maliciously modify either of them.

The state estimation certifier is very restrictive at this stage, but this is acceptable because we only use it on programs synthesized with AUTOFILTER. It only accepts programs written in a generic matrix-assignment based programming language like the one in Figure 3; additionally, those programs are not allowed to redefine variables (except the loop counter) and must consist of exactly one loop which iteratively calculates the best estimate. These programs must be annotated like in Figure 4, i.e., they must start with an annotation containing the specification of the Kalman filter for which the subsequent code is claimed, and then contain lines of code, proof scripts and assertions. All the annotations use directly Maude and/or ITP notation. Each assertion that cannot be proved by straightforward rewriting should come with its proof script annotation. This simple approach works because our synthesized state estimation programs are not concurrent.

The certifier works as follows. It first extends the abstract domain specification with the specification of the program (extracted from the beginning of the program). Then it follows the steps of a proof by mathematical induction on `k`, the loop index. More precisely, it first proof checks the first assertion in the code in which it replaces `k` by 0. Then it incrementally visits each line of code in the loop, adding the assignments to the specification as ordinary MEL equations and proof checking the assertions. In order to proof check an assertion, it calls the ITP tool with the current specification, the assertion, and the proof script provided in the code as annotation. At the end of the loop, it also proof checks the first assertion in the loop in which `k` is replaced by `k+1`.

Therefore, our current certifier simulates the execution of the code modifying its environment (specification) and checking a provided proof whenever an assertion is found.

Assuming that the abstract domain and the Kalman filter are correctly specified, then for any annotated program as above our certifier returns "yes" if and only if the program calculates the best estimate at each iteration. Notice that the certifier was specifically designed to be totally independent from the synthesis engine. The proofs and the annotations are orders of magnitude larger than the real code, but fortunately, they can be automatically generated by the synthesis engine once generic proofs are provided with the program schemas. The ITP proof tasks and their proof scripts generated automatically by our current certifier (about 2300 lines of Maude/ITP) can be seen at `http://ase.arc.nasa.gov/grosu/download/kalman`.

8. Conclusions and Future Work

In this paper, we have shown how to extend program synthesis systems to generate not only code but also proofs of properties of that code. This work was carried out in the context of AUTOFILTER, a synthesizer of state estimation programs, which was augmented to output a proof that the code implements an optimal estimator. This proof is a highly complex proof that cannot be proved automatically but by encoding the key steps of the proof in AUTOFILTER's knowledge base, it is able to generate a proof that can easily be checked by an independent certifier. Such results will encourage the acceptance of code generators in safety-critical domains since the generators will produce not only the code but also certificates that the code is correct.

Currently, our work has focussed on formalizing the domain knowledge required for the certification proofs and on generating the proof for the standard Kalman Filter. However, since AUTOFILTER generates a wide range of variations of Kalman Filter implementations, it must also generate a wide range of proofs. Our hope is that the structure in these proofs is sufficiently similar such that the effort required in formalizing the proofs and encoding them in the knowledge base is manageable. We are currently testing out this hypothesis by considering Kalman Filter variations such as the information filter.

The certifier used in the work described is itself a substantial program, including the Maude system and the ITP tool. Maude and ITP are used both to generate the proofs and to check them. For practical purposes, a certifier must be as simple as possible. Certification authorities will only trust a certification tool if it has been formally verified, and hence it must be small. This means that whilst Maude and ITP are good generic engines for developing domain-specific proofs scripts of individual schemas, the final product will most likely incorporate a kernel certifier with a minimal knowledge base and minimal proving technology.

References

[1] R. Akers, E. Kant, C. Randall, S. Steinberg, and R. Young. Scinapse: A problem-solving environment for partial differential equations. *IEEE Computational Science and Engineering*, 4(3):32–42, 1997.

[2] R. G. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, 3rd edition, 1997.

[3] M. Clavel. ITP tool. Department of Philosophy, University of Navarre, `http://sophia.unav.es/ clavel/itp/`.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, `http://maude.csl.sri.com`.

[5] Compaq. Extended Static Checking for Java, 2000. URL: `www.research.compaq.com/SRC/esc`.

[6] B. Fischer, J. Schumann, and T. Pressburger. Generating data analysis programs from statistical models. In W. Taha, editor, *Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 212–229. Springer, 2000.

[7] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.

[8] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 81–90. IEEE, 2001. Coronado Island, California.

[9] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

[10] J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings, WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.

[11] J. Meseguer and G. Roşu. A total approach to partial algebraic specification. In *International Conference on Automata, Languages and Programming (ICALP'02)*, Lecture Notes in Computer Science, 2002. To appear.

[12] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

[13] PolySpace. URL: `http://www.polyspace.com`.

[14] K. Rustan, M. Leino, and G. Nelson. An extended static checker for modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, April 1998.

[15] Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. In B. Moller, editor, *Mathematics of Program Construction: third international conference, MPC '95*, volume 947 of *Lecture notes in computer science*, Kloster Irsee, Germany, 1995. Springer.

[16] J. Whittle, J. van Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry, and G. Brat. Amphion/NAV: Deductive synthesis of state estimation software. In *Proceedings of Conference on Automated Software Engineering (ASE01)*, San Diego, CA, USA, 2001.